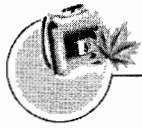

第2章

基础知识

现在我们就可以动手打造这款属于我们自己的操作系统了。当然，在这之前我们还需要将开发操作系统所需的必备基础知识说清楚。众所周知，操作系统是一门偏难的综合性学科。尤其在亲自实践的过程中，对于编程语言、程序算法、硬件原理等相关知识，都需要开发者有相当深入的了解。

因此，无论读者从事的是基于 PC 的程序开发还是基于应用层的嵌入式开发，我们的传统知识结构可能都不足以实现一个完整的操作系统，这可能将会成为我们开发属于自己的操作系统的一个障碍。为了能够扫除这些障碍，我们有必要对一些知识和技巧进行一定程度的补充。

本章正是以此为目的，虽不能对开发操作系统所必备的所有知识进行深入并且全面的讲解，但至少会保证将一些重点或易被忽略的地方拿出来聊一聊。这些内容涵盖了链接和库、基于 ARM 的汇编程序设计、汇编与 C 语言间的混合编程、过程调用标准等。

这些话题都跟程序的编写直接相关，在后续章节的学习过程中都会反复讲到。我们会介绍几个实际的编程例子，确保读者在读完本章之后，至少能够编写并运行一些最基本的程序，为操作系统的开发打下基础。

如果读者对这些知识已有所了解，可以跳过本章，直接学习后边的内容。

2.1 使用 C 语言写第一段程序

我们不如从每个程序员都曾写过的“hello world”切入，来尝试编写能

在虚拟环境中运行的第一段代码，看看操作系统的开发过程究竟是怎样的。

首先请打开文本编辑器，输入以下内容。

代码 2-1

```
#define UFCON0  ((volatile unsigned int *)(0x50000020))
void helloworld(void){
    const char *p="helloworld\n";
    while(*p){
        *UFCON0=*p++;
    };
    while(1);
}
```

程序简单到无须介绍，这里只需要说明一点，物理地址 0x50000020 代表的是 s3c2410 的串口 FIFO 寄存器地址，简单地说，就是写向该地址的数据都将会通过串口发送给另一端。这段程序只是串行的将字符串"helloworld"依次送给串口 FIFO 寄存器。我们将以上内容保存成文件，命名为“helloworld.c”。

接下来让我们尝试编译这段程序，在终端里运行如下命令。

命令 2-1

```
arm-elf-gcc  -O2  -c  helloworld.c
```

编译过程中，我们使用了一个优化参数-O，这样最终生成的代码将由编译器视情况优化，而数字 2 则代表了一个优化级别，数字越高，优化程度越深，但同时带来的不确定性也越多。

目前我们只能将这段代码首先编译生成目标文件，而不能直接生成可执行文件，原因很简单，因为我们的程序当中没有 main 函数。也就是说，编译器找不到程序的运行入口，在链接过程中便会报错。那么是不是说凡是 C 代码，想要生成可执行程序并且成功运行，就必须要有 main 函数呢？

不是。这个答案和我们理解的传统 C 语言程序有些出入，初学者也许不能接受。但事实是 main 函数和普通的函数并没有任何差别，它只不过是由标准所规定的程序入口而已。我们完全可以使用一些手段迫使程序从我们指定的任意函数处开始运行，这一方法可以通过添加适当的编译参数或使用特定的链接脚本来实现。这些内容稍后我们就会讨论。

在命令 2-1 运行之后，一个名为“helloworld.o”的文件便生成了。接下

来需要运行链接工具，用该目标文件来生成一个 ELF 格式的可执行程序。

命令 2-2

```
arm-elf-ld -e helloworld -Ttext 0x0 helloworld.o -o helloworld
```

链接器将目标文件“helloworld.o”链接生成可执行程序 helloworld。参数-e 的作用是指定程序的运行入口。也就是说，可执行程序将会从代码 2-1 中的 helloworld 函数开始执行，取代了默认使用的 main 函数。这样我们就实现了从自定义的函数入口运行程序。参数-Ttext 的作用是指定该程序的运行基地址，此处的值是内存零地址。也就是说，链接工具将目标文件的 hellworld 函数链接到内存 0x0 的位置上，并且从此处执行。

那为什么是内存零地址呢？这是因为在 ARM 体系结构中，程序必须从内存零地址处开始运行。将目标文件链接到内存 0x0 处，我们的 helloworld 理论上就具备了运行条件。

此时，helloworld 程序虽然已经生成，但它仍然不能拿来直接加载到硬件环境里去运行，我们还必须使用如下命令生成一个只包含程序机器码的二进制文件。

命令 2-3

```
arm-elf-objcopy -O binary helloworld helloworld.bin
```

这里的 arm-elf-objcopy 命令将 ELF 格式文件中的二进制机器码抽离出来，生成“helloworld.bin”文件，该文件除了运行时所必要的数据和代码之外，不含有任何其他信息，是纯粹的可执行机器码镜像。而我们原来所理解的可执行文件格式，除了实际代码之外，还会包含 ELF 格式文件头、段头或节头等附加信息。这样程序在运行时，就要首先解析这些附加信息，根据 ELF 格式信息构造出程序运行时所需要的环境，然后才能运行。

接下来我们需要根据程序的编译情况来编写一个 SkyEye 的配置文件，其内容如下。

```
cpu: arm920t
mach: s3c2410x
#physical memory
mem_bank: map = M, type = RW, addr = 0x00000000, size =
0x00800000, file=./helloworld.bin
#all peripherals I/O mapping area
```

```
    mem_bank: map = I, type = RW, addr = 0x48000000, size =
0x20000000
```

在该配置文件中，我们将“helloworld.bin”文件加载到硬件平台 0x0 地址处，与之前编译时程序的链接环境一致。同时，还要保证 0x50000020 处的地址是作为外设寄存器被映射出来的。

将上述内容保存成文件，名为“skyeye.conf”，与“helloworld.bin”文件放置在同一个目录下。在终端运行 skyeye 命令，就会看到如下结果。

```
……
Your elf file is little endian.
arch: arm
cpu info: armv4, arm920t, 41009200, ff00fff0, 2
mach info: name s3c2410x, mach_init addr 0x806ad40
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM   ./helloworld.bin
helloworld
```

可以看出，helloworld 字符串从 SkyEye 虚拟机中打印了出来。我们的第一段 ARM 程序正在运行中！

在上述程序的执行过程中，有一点可能会令读者感到费解。我们的程序明明是将字符串 helloworld 发送给了串口，但为什么最终会在 SkyEye 的页面中出现？

通常在嵌入式产品的开发过程中，串口多是作为设备终端，用于进行产品的调试和操作的。于是 SkyEye 默认地将串口映射到命令行中，以方便虚拟硬件的模拟。当然这也仅仅是虚拟环境所提供的一种方便而已，在实际的串口设备通信中，由于会涉及到传输速率、数据格式等诸多问题，这段程序能够成功运行的可能性并不大。

简要总结一下上面这段程序：我们首先使用编译器将一段代码编译成目标文件，同时又使用链接器将目标文件链接成了可执行程序，并在链接的过程中做了点小动作，使得程序没有使用 main 函数也可以成功编译，又将程序的入口设置在了内存 0x0 处。最后，我们又使用了目标文件复制工具将 ELF 格式的可执行程序转换成只包含代码和数据的原始格式，并通过 SkyEye 成功运行。

在整个过程中，我们涉及了一些编译工具的基本用法。下面我们就来

了解一下链接工具的一种高级用法，那就是通过链接脚本来链接目标文件。

2.2 用脚本链接目标文件

什么是链接脚本？链接脚本就是程序链接时的参考文件，其目的是描述输入文件中各段应该怎样被映射到输出文件，以及程序运行时的内存布局，等等。绝大多数情况下，链接程序在链接的过程中都使用到了链接脚本。

有的读者可能会疑惑，似乎我们在利用 GCC 编译程序的时候并没有使用到什么链接脚本，程序依然会被成功链接并且正常执行。其实，链接工具 arm-elf-ld 在未被指定使用哪一个链接脚本的时候，会使用内嵌到链接工具内部的默认脚本，我们可以使用-verbose 参数来查看该链接脚本。

当应用程序处在操作系统之上时，往往不需要显式指定链接脚本，因为自己编写的链接脚本可能和操作系统默认环境不符，导致运行出错，所以通常使用链接命令内置的脚本，这样可以保证程序运行环境和操作系统默认环境相一致，保证了程序的正常运行。但是，如果我们的程序是运行于操作系统之下或程序就是操作系统本身，那么链接脚本就显得十分重要了。要根据硬件平台的实际环境去编写合适的链接脚本，代码或数据才有可能出现在正确的位置上并正常运行。

本小节我们将讨论链接脚本的具体细节，使用的方式并不是简单地罗列参考手册的内容，而会针对可能用到的知识做深入的研究，将那些根本不可能用到的知识扔到垃圾桶里。

让我们首先来看一下一个典型的链接脚本是什么样子的。

代码 2-2

```
SECTIONS
{
    . = 0x1000;
    .text : { *(.text) }
    . = 0x8000000;
    .data : { *(.data) }
```

```
    .bss : { *(.bss) }
}
```

代码 2-2 就是一个最简单的链接脚本，其中使用了一个非常重要的命令——SECTIONS，这个命令是用来描述输出文件的内存布局的。SECTIONS 命令的标准格式如下。

代码 2-3

```
SECTIONS
{
    sections-command
    sections-command
    ……
}
```

一个 SECTIONS 命令由若干个 sections-command 组成。通过该命令，我们可以设计出程序运行时各段在内存中的分布情况。例如，在代码 2-2 中，代码段被安排到内存的 0x1000 处，数据段则分布在内存的 0x8000000 位置，而 bss 段从形式上紧挨着数据段，整个脚本看起来很容易理解。

SECTIONS 命令的具体用法如下。

（1）点号（.）：点号在 SECTIONS 命令里被称为位置计数器。通俗地讲，它代表了当前位置，你可以对位置计数器赋值，例如，第三行中我们对点号赋予了 0x1000 这个值，其结果是代码段的起始位置从 0x1000 开始。我们也可以将位置计数器的值赋给某个变量，而这个变量可以在源程序中使用。明确指定某个段的起始地址并不是必需的，就如上例中，在 bss 段前面就没有对位置计数器赋值，在这种情况下，位置计数器会采用一个默认值，而这个默认值也会随着各段的大小动态地增加。例如，代码 2-2 中，在数据之前，位置计数器的值为 0x8000000，而在数据段之后，位置计数器并没有被显性赋值，其默认值是 0x8000000 加上数据段的长度。所以，可以说在内存中 bss 段形式上是紧挨着数据段分布的。当然，实际情况可能并非如此，我们稍后会做讨论。如果 SECTIONS 命令一开始就没有对位置计数器赋值，则其默认值为零。

（2）输出段定义：代码 2-2 中，各关键字代表了输出段的段名。.text 关键字定义了该输出位置为代码段，花括号内部定义了代码段的具体内容，其中，星号（*）代表所有文件，*(.text)的意思是所有目标文件的代码段都

将被链接到这一区域，我们也可以特别地指定某个目标文件出现在代码段的最前面。.data 关键字定义了该输出位置为数据段，其格式和用法与.text 关键字相同。.bss 关键字定义了输出位置是 bss 段，格式和用法与上面相同。这种写法很可能给我们造成一种错觉，即对输出段的定义必须以 text、data 或 bss 关键字命名。其实输出段完全可以任意定义，因为输出段的实际内容与输出段的命名无关，而只与花括号内的具体内容有关。

（3）ALIGN(N)：在代码 2-2 中并没有 ALIGN 关键字。但是在实际应用中，我们经常需要使用 ALIGN 产生对齐的代码或数据。很多体系结构对对齐的代码或数据有严格的要求，还有一些体系结构在处理对齐的数据或代码时效率更高，这都体现了 ALIGN 关键字的重要性。我们可以用某一数值取代 ALIGN()括号中的 N，同时对位置计数器赋值，如.=ALIGN(4)就表示位置计数器会向高地址方向取最近的 4 字节的整数倍。

链接脚本除了 SECTIONS 命令之外，还有一个比较常用的命令是 ENTRY 命令，该命令等同于 arm-elf-ld 命令的参数-e，即显式地指定哪一个函数为程序的入口。

现在我们可以应用上面提到的知识，针对前面的 helloworld 程序写一个链接脚本，其内容如下。

代码 2-4

```
ENTRY(helloworld)
SECTIONS
{
    . = 0x00000000;
    .text :{
        *(.text)
    }
    . = ALIGN(32);
    .data :{
        *(.data)
    }
    . = ALIGN(32);
    .bss :{
        *(.bss)
    }
}
```

结合前面的描述，上面这段链接脚本的作用是将程序的代码段链接到地址 0x0 处，保证了系统上电时就能从该地址中获取第一条指令并运行。程序的数据段在代码段之后，而 bss 段则被安排到数据段之后。这两个段的起始地址都是经过对齐的。最后，链接脚本还明确规定将名为 helloworld 的函数作为整个程序的入口。这也就是说，helloworld 函数就是出现在内存 0x0 位置上的那个函数。

好了，链接脚本写完了，那我们应该如何使用该脚本呢？这就要在运行链接命令时借助于-T 参数。使用-T 参数可以让链接器从脚本中读取并解析相应命令，然后按照命令的要求链接生成可执行程序。

如果针对前面的例子，它的用法会像下面这样。

命令 2-4

```
arm-elf-ld  -T  helloworld.lds  helloworld.o  -o  helloworld
```

这样我们就得到与前一小节同样的结果，但使用的方法却更加专业。

2.3　用汇编语言编写程序

前面能够运行在虚拟硬件环境中的第一段程序是使用 C 语言写成的。理论上我们可以完全使用 C 语言来编写整个操作系统。但在实际应用中，完全使用 C 语言编写的操作系统却寥寥无几。汇编语言虽然有很多的缺点，但在操作系统底层开发中，有时却能发挥出不可替代的作用，这一点相信读者会在今后的学习中有深入的体会。正因为如此，我们还需要利用一节的篇幅，说一说如何使用汇编语言进行 ARM 程序开发。

我们仍将采用边做边讲、在遇到问题的时候再去说明的方式，避免因为枯燥地将知识点罗列出来而影响读者的兴趣。

下面仍从一个例子开始。

代码 2-5

```
.arch armv4
.global helloworld

.equ REG_FIFO,  0x50000020
```

```
.text
.align 2

helloworld:
    ldr r1,=REG_FIFO
    adr r0,.L0
.L2:
    ldrb r2,[r0],#0x1
    str r2,[r1]
    cmp  r2,#0x0
    bne .L2
.L1:
    b   .L1
.align 2
.L0:
    .ascii   "helloworld\n\0"
```

代码 2-5 就是 helloworld 程序的汇编版本。想要让读者理解这段程序，我们并不需要对 ARM 的汇编程序设计进行太过系统的介绍，我们只需要说清楚两点，一是寄存器，二是指令集。

寄存器是CPU进行运算时所必需的存储设备。ARM上的寄存器很丰富，能够参与普通运算的从 R0 到 R12，就有 13 个之多。如果没有特殊规定，程序在运算时可以随意选择 R0~R12 的任意一个寄存器参与运算。除了这些寄存器之外，在 ARM 体系结构中还包括负责保存堆栈地址的 R13 寄存器，负责保存程序返回值的 R14 寄存器以及负责记录程序地址的 R15 寄存器。这些寄存器由于拥有专门的功能，所以有的时候也使用别名来称呼它们，分别叫做 SP、LR 和 PC。另外，ARM 还有一个非常特殊的寄存器专门用来保存程序的运行状态，叫做程序状态寄存器，使用 CPSR 来表示。CPSR 中既包含描述程序在运算过程中是否产生了溢出、负数等状态的相应位，也包含用于描述当前处理模式的模式位。关于处理器模式，因为代码 2-5 中没有涉及，所以我们会在它们出现的时候进行详细讲述。

说完了寄存器，我们再来聊一聊指令。在汇编程序设计过程中，指令是我们构成汇编程序的基本单元。

指令包括指令助记符和伪指令。每一条指令助记符都代表 CPU 提供的某一条指令，也可以说指令助记符都唯一地对应了 CPU 的一条机器码。因

为机器码本身就是无规则的二进制数，本身不容易被记忆和使用，所以用助记符来帮助记忆，这其实就是“助记符”三个字的真正意思。

伪指令不像指令助记符那样是在程序运行期间由 CPU 来执行的，通常伪指令只会作用在编译过程中，对最终生成的文件造成不同程度的影响。有些伪指令在编译的时候并不生成代码，还有些伪指令则会扩展成一条或多条实际的指令。

下面我们就对代码 2-5 中所使用的指令和伪指令做深入的介绍。

（1）.arch 伪指令：该伪指令的作用是选择目标体系结构。例如，在代码 2-5 中，.arch 表示该段汇编代码将会被编译生成符合 armv4 体系结构的代码，从而实现了为特定平台生成特定的代码。如果不想使用该伪指令，那么在程序编译的时候，使用-march 参数也能达到同样的效果。

（2）.global 伪指令：该伪指令的含义是让 global 过的符号对链接器可见，也就是说，一个函数或变量，通常情况下只在本文件内有效，当需要在外部引用该文件里的某一个函数或变量时，必须首先将该函数或变量使用.global 伪指令进行声明。在代码 2-5 中，helloworld 函数作为我们程序的入口函数，必须在链接时用-e 参数来指定，或者在链接脚本中用 ENTRY 命令来做显示声明。因此.global 伪指令在这里就显得至关重要了。

（3）.equ 伪指令：该伪指令其实很简单，相当于 C 中的宏定义。在代码 2-5 中，正是使用了.equ 伪指令将 0x50000020 用宏 REG_FIFO 来代替。

（4）.text 伪指令表示从当前位置开始的内容被归并到代码段中。

（5）.align 伪指令：我们并不是第一次遇到 align 这个词，回想一下链接脚本一节，关键字 ALIGN 的作用是在链接时，迫使被修饰的内容对齐。这里的.align 与 ALIGN 关键字意义相同，也能够更新位置计数器的值，使代码对齐到某一边界。但此处需要强调的是，该伪指令针对 ARM 汇编的用法与其他体系结构稍有不同。例如，在 m68k、sparc 以及运行有 ELF 文件格式的 x86 结构中，.align 后边的数字直接代表了要对齐的字节数，比如.align 8 表示此处代码会按照 8 字节边界对齐，而在 ARM 体系结构下，.align 后边的数是以幂的形式出现的，正如代码 2-5 中那样，.align 2 表示此处是以 4 字节对齐的。

（6）.ascii 伪指令：该伪指令用于在内存中定义字符串，我们要输出到串口中的字符串“helloworld\n”就是由它定义的。

（7）ldr 伪指令：很多朋友在看了前面的介绍后可能会得到一个错误结论，即所有伪指令都是以点开头的。其实不然，ldr 就是一个反例。ldr 被称为常量装载伪指令，其作用是将一个常量装载到寄存器中。因为 ARM 指令等宽指令格式的限制，不能保证所有的常数都可以通过一条指令装载到寄存器中。程序编译时，如果能够将 ldr 指令展开成一条常量装载指令，则编译器就会用该指令代替 ldr，否则编译器会首先开辟一段空间存储被装载的常量，然后使用一条存储器读取指令将该常量读入到寄存器当中。

（8）adr 伪指令：adr 伪指令被称做地址装载伪指令，与 ldr 类似，adr 伪指令能够将一个相对地址写入寄存器中。

至此，代码 2-5 中使用的全部伪指令我们就介绍完了。这里我们想强调的是，伪指令都是与编译器有关的，换句话说，即使是同一种硬件平台，GCC 中的伪指令与其他编译器中的伪指令也并不完全兼容。

我们前面说过，伪指令要么作用在编译过程中对最终生成的文件造成影响，要么会扩展成一条或多条实际的指令，这些都是在程序编译过程中由编译器决定的，因此，不同的编译器的伪指令集可能不同。而指令是与硬件平台有关的，即使不同的编译器对同一指令使用的助记符不一致，它们生成的机器码仍然是一致的，并且不同编译器所使用的指令助记符集几乎都彼此相同的，因为指令助记符都是由芯片生产厂商定义的，编译器大多会遵守这种定义。

说完了伪指令，我们来介绍一下代码 2-5 中的指令。

（1）内存装载指令 ldr：ldr 如果作为实际指令出现，表示从内存读取数据到寄存器中，而代码中使用了一个 b 作为后缀，表示读取的只是一个字节的数据。相信读者能够举一反三，明白 ldrh 指令的含义。在该语句中，[r0] 表示地址值，而后边的#0x1 是常数 1，代表寄存器 R0 的增量，这种寻址方式被称为立即数后变址寻址。整条语句的意思是，首先从寄存器 R0 所指向的地址中读取一个字节的数据存储在 R2 中，然后 R0 中的值自加 1。因为在这之前 R0 的值已经指向了 helloworld 字符串的首地址，所以这条语句第一次执行，就会把字符‘h’存储在 R2 中。

（2）内存存储指令 str：str 能够将寄存器中的值存储到另一个寄存器所指向的内存地址中。代码 2-5 正是使用了 str 指令将刚刚读取到的 R2 中的值存储到 R1 地址处，而该地址正是串口 FIFO 寄存器的地址。因此，字符

‘h’就会被显示到屏幕当中。

（3）比较指令 cmp：比较指令是将待比较的两个数相减，然后去影响标志位，所以它实际上是一条不返回运算结果的减法指令。比较指令不保存结果，但是会使当前程序状态寄存器 CPSR 中零标志位置位或清除，进而影响条件判断语句的执行。

（4）跳转指令 b：接下来我们看到的指令 bne 其实就是跳转指令 b，后缀 ne 是条件码，表示执行条件为“不相等”。整个语句可以解释为如果不相等，则跳转到符号.L2 处。这里的不相等指的正是上一条指令比较产生的结果。因为比较指令只影响标志位，所以我们可以说条件码只是根据相应标志位的值来有选择地执行。

程序在每次向串口 FIFO 寄存器写入数据后，都要与零进行比较。当比较的结果不相等，表示字符串还没有结尾，则继续将下一个字符写入到串口 FIFO 寄存器中；如果字符与零相等，证明读到了字符串结束符，然后程序进入死循环，这就是程序 2-5 的执行过程。

最后我们可以使用与代码 2-1 同样的编译方法来编译这段汇编版的 helloworld。

总结一下，现在我们已经学习了 ldr、str、cmp 和 b 这四条汇编指令，也学习了 arch、global、equ、text、align、ascii、ldr 和 adr 这八条伪指令。这些指令不但常用，而且很重要。在操作系统代码的编写过程中，也会被经常使用。

2.4　汇编和 C 的混合编程

很少有一款操作系统会完全使用汇编语言来编写，也几乎没有一个操作系统是完全使用 C 语言写成的。汇编语言负责编写启动、堆栈初始化、系统运行段划分等关键部分的代码，而 C 语言则被用在涉及到算法、逻辑等与硬件关联不大的地方，解决复杂的问题。因此需要将两者完美的结合，才能发挥出各自的优势。

这一节我们就来探讨一下 ARM 的混合编程问题，为我们以后正式编写

操作系统打下基础。

通俗地讲，无论是汇编语言、C 或是其他类型的语言，都是人能够理解的语言，都有特定的语法规则支持各种执行逻辑。从这一点出发，所有的语言（包括人类互相交流的语言），彼此之间都没有什么不同。而编译生成的机器码只有机器才能理解，并且不同的硬件平台彼此又语言不通，这样，问题似乎就来了，人能理解的语言机器理解不了，机器能理解的语言人又很难掌握，这个问题该怎样解决呢？

编译器能够帮助我们解决这个问题，编译器的作用就是将人能够理解的语言翻译成机器能够理解的语言，如图 2-1 所示。从这个角度看，把编译器叫做翻译器，多少也有些道理。

既然人能理解的语言五花八门，而机器能理解的语言对于特定平台又是唯一的，因而不同语言间的混合编程，就要从硬件的角度去理解才会有意义。

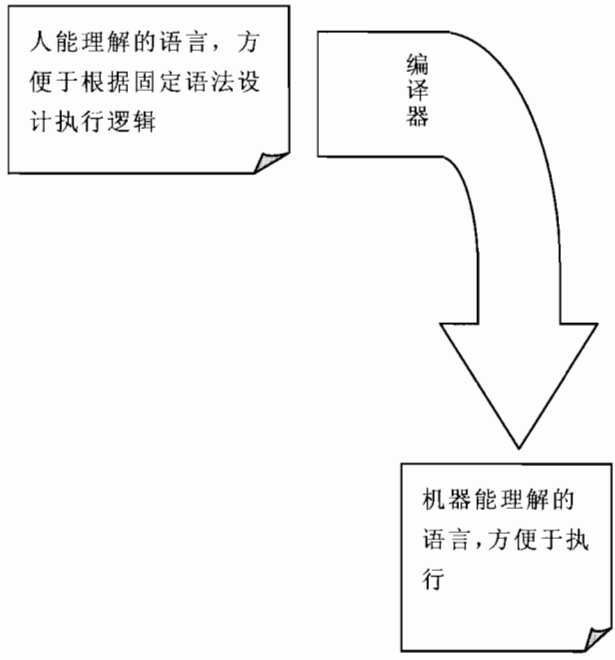

图 2-1　编译器的作用

2.4.1　过程调用标准

要想具体实现汇编语言和 C 语言之间的混合编程，就必须制定出一套

统一的标准并约束双方共同遵守，我们将这套共同遵守的标准称为过程调用标准（Procedure Call Standard），简称为 PCS。

过程调用标准其实就是一套规定，任何编程语言在应用的时候，只要去遵守这种约定，就能够与同样遵守该约定的其他编程语言和平共处。因此，想要让 C 语言和汇编语言能够混合编程并且正确执行，对过程调用标准的严格遵守就显得相当重要了。

其实，过程调用标准并没有具体针对任何语言，也就是说，无论什么编程语言，只要遵循这种标准，就可以直接调用其他同样遵守该标准的编程语言写出来的程序，而不一定非要限制在 C 语言或汇编语言上。ARM 的 PCS 有很多版本，如 ARM 过程调用标准 APCS、Thumb 过程调用标准 TPCS，以及 ARM-Thumb 之间互相调用时所要遵守的过程调用标准 ATPCS。目前“最新”的一套标准名为 ARM 体系结构过程调用标准（Procedure Call Standard for the ARM Architecture），简称 AAPCS，该标准定义了 ARM 下的若干个子例程如何才能实现独立编写、独立编译、独立汇编，却可以共同运行。

我们依旧会借助一个实际例子来介绍一下一些基本的过程调用规则，看看如何实现 C 语言和汇编语言的混合编程。

2.4.2　混合编程的例子

代码 2-6

```
.arch armv4
.global _start

.equ REG_FIFO,  0x50000020

.text
.align 2

_start:
    ldr r0,=REG_FIFO
    adr r1,.L0
    bl helloworld
.L1:
```

```
    b   .L1

.align 2
.L0:
    .ascii  "helloworld\n\0"
```

代码 2-6 是一段汇编程序，它与代码 2-5 很类似。二者的区别在于代码 2-6 不是直接将 helloworld 字符串送给串口 FIFO 寄存器，而是通过 bl 指令调用 helloworld 函数，由该函数负责数据写入。

bl 指令我们没有接触过，它的功能与 b 指令几乎一致。唯一的不同是 bl 指令在运行时，能够自动地保存程序的返回地址。这样，在成功调用子程序并且运行结束时，就能够返回原先的位置继续运行。

helloworld 函数是由 C 语言写成的。根据 AAPCS 的规定，当函数发生调用的时候，函数的参数会保存在 ARM 核心寄存器 R0~R3 中，如果参数多于 4 个，则剩下的参数会保存在堆栈中。因此，我们也会把寄存器 R0~R3 称为参数寄存器（argument register），用别名 a1~a4 代替。在代码 2-6 中，调用 helloworld 函数之前，我们分别对 R0、R1 两个寄存器赋值，这两个值就是 helloworld 函数的参数。

下面就让我们来了解一下这个函数具体是怎样实现的。

代码 2-7

```
int helloworld(unsigned int *addr,const char *p){
    while(*p){
        *addr=*p++;
    };
    return 0;
}
```

代码 2-7 类似于之前用 C 语言实现的 helloworld 函数。不同的是该函数多了两个参数，分别是内存地址和要写向该地址的字符串的首地址。

整个函数的功能是将指针 p 所指向的字符串写向由 addr 代表的内存地址中。当函数 helloworld 执行完毕后，程序会返回到代码 2-6 的.L1 地址处继续运行。helloworld 函数带有一个 int 型返回值，该返回值会保存到核心寄存器 R0 中，如果返回的是一个 64 位的值，则该值会由 R0、R1 同时保存。很显然这也是由 AAPCS 所规定的。

这两段程序在编译的时候会稍有不同，首先我们必须分别编译这两段程

序，产生两个目标文件。

将代码 2-6 命名为“start.s”、代码 2-7 命名为“helloworld.c”。接着我们要使用类似于命令 2-1 的命令来生成两个目标文件。然后，使用命令 2-5 来编译生成 ELF 格式的可执行文件。

命令 2-5

```
arm-elf-ld -e _start -Ttext 0x0 start.o helloworld.o -o helloworld
```

在终端运行 skyeye 命令，我们可以得到与代码 2-1 相同的结果。这表示我们已经成功实现了 C 和汇编的混合编程。

2.5 Makefile

在这一小节中，我们将介绍一种有效的编译手段来提高代码的编译效率。

我们看到，在前面的所有例子中，程序都是采用一种“单打独斗”的编译方法来编译的。程序在编写完成后，需要程序员手动运行命令去编译链接，最终生成可以运行的程序。当然，如果只是编译一两个文件，这似乎并不是什么问题。但一旦程序的规模扩大，需要编译的文件增多，程序编译过程就将会给我们带来极大的负担。

因此，这样的编译方式很显然不合适对操作系统进行。幸好 make 工具可以帮助我们解决所有的问题。

make 工具能够自动地按照我们的要求依次编译指定的源程序，在项目工程中是一个不可或缺的工具。make 工具功能非常强大，而在使用 make 工具的同时，我们还需要写一个 Makefile 文件。顾名思义，Makefile 文件就是专门供 make 命令使用的文件，里面记录的都是编译源代码的具体细节。这里我们给出了相对通用的 Makefile 文件模板，内容如下。

代码 2-8

```
CC=arm-elf-gcc
LD=arm-elf-ld
OBJCOPY=arm-elf-objcopy
```

```
CFLAGS= -O2 -g
ASFLAGS= -O2 -g
LDFLAGS=-Tleeos.lds -Ttext 30000000

OBJS=start.o helloworld.o

.c.o:
    $(CC) $(CFLAGS) -c $<
.s.o:
    $(CC) $(ASFLAGS) -c $<

leeos:$(OBJS)
    $(CC) -static -nostartfiles -nostdlib $(LDFLAGS) $? -o $@
-lgcc
    $(OBJCOPY) -O binary $@ leeos.bin

clean:
    rm *.o leeos leeos.bin -f
```

对于不熟悉 Linux 或 Makefile 的读者来说，这段代码就像天书一样。这里我们不会深入研究 Makefile 的写法，只需要知道其用法即可。毕竟这些知识离操作系统的原理还是比较远的。

代码 2-8 可以作为一个模板，只需要稍做修改就能使用。具体的方法是，如果我们需要添加并编译新的源文件，只需要在变量 OBJ 中增加与源文件同名的目标文件即可，仅此而已，这个文件就可以用来编译我们自己的操作系统了，其他参数无须修改。

如果读者想要在其他场合下使用该模板，那么当需要修改编译选项时，可以将其添加到变量 CFLAGS 或 ASFLAGS 中，LDFLAGS 变量代表链接选项，同样可以根据实际情况更改。CC、LD、OBJCOPY 则代表编译工具，也可以自行确定。

这里需要强调的是，在编译的过程中，除-nostdlib 之外，我们使用了另一个不太常用的 GCC 编译选项，那就是-nostartfiles。简单地说，这个选项的作用就是不允许编译器使用默认的启动文件。因为我们要写的是一个操作系统，系统的启动是自行处理的，不需要编译器帮我们添加。同时，在程序中我们通过-lgcc 在编译时链接 libgcc.a 这一函数库，这样，ARM 中的一些基本运算（如除法）就不需要我们在操作系统中亲自实现了。

将代码 2-8 保存到名为 Makefile 的文件中。此时我们便可以进行编译了，确保所有的文件都在同一个目录下，打开一个终端，切换到源代码所在目录，同时运行如下命令。

命令 2-6

```
make
```

这样就会得到与上一节同样的结果，所付出的仅仅是适当修改 Makefile 文件中的 OBJS 变量的内容，并在终端敲四个字母。

如果想要同时删除所有编译生成的文件，只保留程序代码，可以通过如下命令来实现。

命令 2-7

```
make clean
```

2.6　总结

目前，我们完成的工作并不多，但至少为整个系统的完成提供了一个结构或者说一种解决的方法。在以后的工作中，我们无非也是利用类似的方法和技巧，结合我们对操作系统的理解，一步一步地实现属于我们自己的嵌入式操作系统！